ECE 364 Prelab Assignment 06

Regular Expressions

September 24, 2018

Passing this lab will satisfy course objectives CO2, CO3, CO5

In order to pass CO5, you must use regular expressions searching or matching to solve this lab. Using any other method, like string functions, or the `split()` function `str.split()` or `re.split()`, will *not* earn you any credit, nor will it satisfy the course objective. If you are in doubt, check with your TAs.

Instructions

- You must meet all *base requirements* in the syllabus to receive any credit.
- Follow all rules in the **Code Quality Standards**.
- Work in your Prelab06 directory.
- Copy all files from ~ee364/labfiles/Prelab06 into your Prelab06 directory. You may use the following command:
  `cp -r ~ee364/labfiles/Prelab06/* ./`
- To submit, commit your files to SVN. We will only grade the version of the file that is in SVN.
- Make sure your output from all functions match the given examples. **You will not receive points for a question whose output mismatches the expected result.** For functions that return an unordered collection (e.g., `dict`, `set`, etc.), it's okay if the order in your output differs from what is shown in this document. They are still considered "equal".
- Unless otherwise specified, you may not use any external library. You may use any module in the **Python Standard Library** to solve this lab.
- Make sure you are using Python 3.4 for your project. In PyCharm, go to:

  File Menu → Settings → Project Interpreter

  And make sure that Python 3.4 (`/usr/local/bin/python3.4`) is selected.

Failure to meet all base requirements will result in zero credit. This includes submitting code that works with other versions of Python, but not Python 3.4.

Tip: Before you begin, read the slides about "greedy" vs. "non-greedy" matches in the lecture notes.

Regular Expressions

Create a Python file named `regex_app.py`, and do all of your work in that file. This file should only consists of function blocks, and, optionally, a conditional main block, (i.e. `if __name__ == "__main__":`). Do not include any loose Python statements, but you can, however, write any number of additional utility/helper functions that you might need.

Part I

- **[10 pts]** One possible format of a URL is:

  `http://[base_address]/[controller]/[action]?[query_string]`

  where `[query_string]` contains a list of `field=value` elements, separated by the ampersand (&) symbol. For example:

  `http://www.purdue.edu/Home/Calendar?Year=2018&Month=September&Semester=Fall`

  Note that all elements of the URL can only contain alphanumeric characters, the underscore (_), the dash (-) or the dot (.). Write a function called `get_url_parts(url)` that takes in a URL of the above format, and returns a `(string, string, string)` tuple, where the elements of the tuple are the base address, the controller and the action. For example:

```
>>> url = "http://www.purdue.edu/Home/Calendar?Year=2018&Month=September&Semester=Fall"
>>> get_url_parts(url)
("www.purdue.edu", "Home", "Calendar")
```

- **[10 pts]** Following the information in the previous question, write a function called `get_query_parameters(url)` that takes in a URL of the above format, and returns a list of `(string, string)` tuples, where the elements of the tuple are the field and the value of each query element. For example:

```
>>> url = "http://www.google.com/Math/Const?Pi=3.14&Max_Int=65536&What_Else=Not-Here"
>>> get_query_parameters(url)
[("Pi", "3.14"), ("Max_Int" ,"65536"), ("What_Else", "Not-Here")]
```

- **[10 pts]** Write a function called `get_special(sentence, letter)` that takes in a sentence and a single letter, and returns a list of the words that start or end with this letter, but not both, regardless of the letter case. For example:

```
>>> s = "The TART program runs on Tuesdays and Thursdays, but it does not start until
next week."
>>> get_special(s, "t")
["The", "Tuesdays", "Thursdays", "but", "it", "not", "start", "next"]
```

- **[10 pts]** Write a function called `get_real_mac(sentence)` that takes in a string and searches for the presence of a MAC address anywhere in the sentence. If found, return it as a string. Otherwise, return `None`. An example of a MAC address is `58:1C:0A:6E:39:4D`.

  Notes:

  - The MAC address is comprised of six parts, each part consists of two hexadecimal digits.
  - The parts are separated by a colon, ':', as shown above, or by a dash, '-' as in `58-1C-0A-6E-39-4D`.
  - The letters in the hexadecimal digits can be present in uppercase or lowercase form.
  - The MAC address can be present anywhere in the sentence.

Part II

You are given a data file called `"Employees.txt"` containing employee information for some company, where each line represents an employee entry. The information required for every employee are: "name", "ID", "phone number" and "State," which is the US State name in which the employee resides. Unfortunately, the information in the file is not properly maintained, and hence not all the required employee information is present in the file. Moreover, for the same piece of information, the format is not consistent. The description of the data contained in the file is as follows:

- The employee name is always present, but it is sometimes formatted as `"<First> <Last>"`, and other times as `"<Last>, <First>"`. Both the first and last names only contain uppercase and lowercase letters.
- The employee ID is a UUID[1], a 32 hexadecimal character string. The canonical form of a UUID is formatted as 8-4-4-4-12, e.g. `4f43b41b-c200-4c09-9d45-8892c5b70cea`. However, the ID in the file can be present in lowercase or uppercase form. It also may or may not be hyphenated, and it may be surrounded by curly brackets. Here are some examples of how the given ID might be present:

  ```
  4f43b41b-c200-4c09-9d45-8892c5b70cea
  {4f43b41b-c200-4c09-9d45-8892c5b70cea}
  4F43B41B-C200-4C09-9D45-8892C5B70CEA
  4F43B41BC2004C099D458892C5B70CEA
  {4F43B41B-C200-4C09-9D45-8892C5B70CEA}
  ```

  Finally, note that the ID may be missing from the employee entry.
- The employee phone number can be formatted as one of the following: `"(XXX) XXX-XXXX"`, `"XXX-XXX-XXXX"` or `"XXXXXXXXXX"`. The phone number may be missing from the entry.
- The State contains only letters, but it can be one word, like "Indiana" or two words, like "New York." The State name may be missing from the entry.
- For every entry, the fields are separated by a varying number of semicolons (;), commas (,) and spaces.
- The order of the fields is always the same: `name, ID, phone, state,` i.e. if the ID and the State are present, the ID always shows up before the State.
- If an entry has all of the fields, i.e. `name, ID, phone, state,` present, this entry is considered complete.
- If an entry has only the employee name present, but no other fields, that entry is rejected.
- If an entry has the employee name and one or two additional fields, that entry is considered partially complete.

Your task is to create a set of functions to parse the file and return the required data in a normalized format.

Note: It is possible to create a single regular expression pattern to parse through all the lines at once. However, this is not required, nor recommended for this lab unless you are comfortable enough with Regular Expressions.

[1] UUID stands for **"Universally Unique Identifier"**. This is also sometimes referred to as a GUID, which stands for **"Globally Unique Identifier"**.

Requirements

- In all of the functions below, when returning an employee name, it must be formatted as `"<First> <Last>"`.
- If any function returns a phone number, it must be formatted as `"(XXX) XXX-XXXX"`.
- If any function returns an ID, it must be formatted in the canonical form of 8-4-4-4-12, where all the characters are in lowercase. This can be achieved by using the function `UUID()` from the `uuid` module. You will need to use the string representation of the UUID, by using the `str()` to get the desired canonical form. For example:

```
>>> from uuid import UUID
>>> i = "{4F43B41BC2004C099D458892C5B70CEA}"
>>> str(UUID(i))
"4f43b41b-c200-4c09-9d45-8892c5b70cea"
```

- [**10 pts**] Write a function called `get_rejected_entries()` that returns a sorted `list` of the rejected employee names, i.e. the employees whose names are present, but with no other information.
- [**10 pts**] Write a function called `get_employees_with_ids()` that returns a `{string: string}` dictionary, of all employees with IDs, where the key is the employee name, and the value is the ID. The dictionary must contain the employees whose IDs are present in the file, regardless whether any other information is present or not.
- [**10 pts**] Write a function called `get_employees_without_ids()` that returns a sorted `list` of employee names of the "non-rejected" employees whose IDs are *not* present. (Note that these employees must have phone and/or State information.)
- [**10 pts**] Write a function called `get_employees_with_phones()` that returns a `{string: string}` dictionary, of all employees with phone numbers, where the key is the employee name, and the value is the phone number. The dictionary must contain the employees whose phone numbers are present in the file, regardless whether any other information is present or not.
- [**10 pts**] Write a function called `get_employees_with_states()` that returns a `{string: string}` dictionary, of all employees whose State of residence is present, where the key is the employee name, and the value is the State name. The dictionary must contain the employees whose State of residence is present in the file, regardless whether any other information is present or not.
- [**10 pts**] Write a function called `get_complete_entries()` that returns a dictionary, where the key is the employee name, and the value is the three-element tuples `(string, string, string)`, where the first element is the ID, the second is the phone number, the third is the State of residence. This dictionary must contain the employees whose information is complete in the file, i.e. their IDs, phone numbers and States are all present.